The Software Toolworks
Walt Bilofsky, Prop.

14478 GLORIETTA DRIVE
SHERMAN OAKS, CALIFORNIA 91423

TELEPHONE
(213) 986-4885

C/80
Version 2.0
February 1982
Walt Bilofsky

Table of Contents

1. INTRODUCTION

C/80 is a compiler for the C programming language, running under the CP/M and HDOS operating systems. It requires a minimum of 48K of memory. The compiler produces an assembly language text file which is turned into an executable object program by an absolute assembler which is included (except on HDOS, where ASM is used). Optionally, C/80 can produce output for Microsoft's Macro-80 relocatable assembler.

The reference manual for C/80 is The C Programming Language by Brian Kernighan and Dennis Ritchie. Section 2 tells where you can obtain this book.

Purchasers of inexpensive C compilers have come to expect that they will lack many important language features, or will have non-standard variations which make C programs less portable. C/80 Version 2.0 is one of the better compilers in this respect. It supports all of the language features described in The C Programming Language, with the following exceptions:

- **float**, **double**, and **long** data types
- **typedef**
- Arguments to **#define** macros
- Bit fields
- **#line**
- Declarations within nested blocks

C/80 Version 2.0 does support structures, statics, initialization, casts, compile time evaluation of constant expressions -- in short, all other C language features. A few language features have restrictions on their use; see Section 8 for a complete list of the exceptions and implementation dependencies.

This C implementation also provides the following features:

- Runtime command arguments including I/O redirection.
- Conventional C I/O library
- Random access file I/O
- Dynamic storage allocation
- Runtime execution profile facility
- Selectable Macro-80 compatibility
- In-line assembly language
- Incudes absolute assembler (CP/M only)

The objective of Ron Cain's small-C implementation was to make a subset of the C language available to the computer hobbyist at minimal cost. We have continued in that spirit by keeping the price of C/80 as low as possible. However, we have dedicated a considerable amount of work and compiler expertise to developing C/80, which now presents both the beginner and the serious programmer with a genuinely useful tool for program development. Many products from The Software Toolworks are written in C/80, including TEXT, LISP/80, UVMAC, ED-A-SKETCH, and C/80 itself.

Acknowledgements:

Ron Cain's contribution in providing a simple, public domain compiler for a minimal C subset is well known and widely appreciated. Jim Gillogly wrote and maintained **printf**. The initial version of **seek** and HDOS **exec** were contributed by Al Bolduc. CP/M **exec** was written by Robert Wesson.

Note: This document describes C/80 implementations for both CP/M and HDOS. Where program names, devices, etc., differ for the two systems, the CP/M names will be used, with the HDOS equivalent in brackets, [like this].

2. FOR THE NEW C PROGRAMMER

A summary and brief description of the C/80 language appears below (Section 7). For a detailed introduction to C, the beginner should obtain The C Programming Language, by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). This book defines the C language, and contains many useful programs as examples. It is sold in many computer stores and college and technical bookstores, or you may purchase it by mail from either of the following:

Lifeboat Associates
1651 Third Avenue
New York, NY 10028
(212) 860-0300; phone orders $50 minimum.
Price: $14.00 + $3.93 shipping (UPS Blue in US)
MC, VISA, AE accepted. C.O.D: add $5.00.

Prentice-Hall
200 Old Tappan Road
Old Tappan, NJ 07675
(201) 767-5067

Note: In reading The C Programming Language, it is important to keep in mind at least the major features of C which are not supported by C/80. Section 8 below lists the differences between C/80 and the language described in the book.

3. FOR THE EXPERIENCED C USER

Users who know C will find C/80 quite familiar and easy to use. It supports data types **char** and **int**, and full C pointers, arrays and structures. All C control statements are supported, all operators, and most preprocessor functions. The preprocessor allows in-line assembly language code.

C/80 programs use the conventional command line (**main(argc,argv)**). The runtime library provides many of the capabilities of the standard C I/O library file handling routines, and implements I/O redirection in the command line.

There are a few differences between C/80 and full C, notably the absence of **long** and **float**. The one difference which may cause hard to detect problems in converting full C programs is that functions may not be called with a different number of arguments than specified in the function definition.

C/80 program source files are normally prepared using the full upper and lower case ASCII character set. Users with upper case only terminals should refer to Section 14.

4. THE C/80 DISTRIBUTION DISK

The C/80 Distribution Disk contains the following files. A few of the files are not included on the Osborne 1 due to lack of disk space.

C.COM or C.ABS — The C/80 compiler.

CLIBRARY.ASM — The C/80 I/O library. This file is automatically included in all C/80 program assemblies. It must reside on A: [SY0: on HDOS] at assembly time.

CLIBRARY.REL — A relocatable version of the library, for use with Microsoft's Link-80

SEEK.C — Routines for random access file I/O (not for CP/M 1.4 and earlier).

EXEC.ASM — [HDOS: EXEC.C]. Routine to chain to another .COM [.ABS] program. (Not included on Osborne.)

CPROF.ASM — The runtime execution profile library. This file is automatically included in assemblies of C/80 programs compiled with the -p switch (see Section 12). It must reside on A: [SY0: on HDOS] when such programs are assembled. (Not included on Osborne; generate from CPROF.C.)

CPROF.C	Source file for CPROF.ASM. Provided to allow alteration of runtime profile to print in other formats, accumulate different data, etc. Osborne users must generate CPROF.ASM by compiling this file.
CTRACE.ASM	An alternate runtime execution profile library. Instead of printing a runtime profile, CTRACE traces each routine call and return. See Section 12.
PRINTF.C	The C80 formatted output routines (see Section 9). May be incorporated into a program using **#include printf.c"**, which should appear before any use of **printf.**
PRINTF.H	A header file for defining **printf** in source files where the entire **printf.c** source file must not be included.
HELLO.C	A sample C/80 program; the first program in The C Programming Language.
TAB.C	A sample C/80 program which copies a file, replacing blanks by tabs wherever this might result in a savings of space. (Not included on Osborne.)
CMP.C	A sample C/80 program which compares two files.
TREE.C	A sample C/80 program showing the use of structures.
UCASE.C	A sample upper case C/80 source file. UCASE translates C/80 source files prepared in lower case into upper case format. (Not included on Osborne.)
AS.COM	(CP/M only.) An absolute 8080 assembler which assembles C/80 output files. This assembler is essentially equivalent to the ASM assembler under HDOS. Section 13 gives a brief description of how to use **AS.**
PATCHES.DOC	The default values and patch locations for the sizes of various compiler tables. See Section 6.

5. AN EXAMPLE

This section describes how to assemble and run a C/80 program. The example shown is for the CP/M operating system. [Under HDOS, the procedure is identical, except that HDOS uses the ">" prompt instead of "A>", and the Heath assembler ASM is used, with the command "asm hello=hello".]

First a source file, called HELLO.C, must be prepared. This can be done using any text editor. The program on the source file should look like this:

```
#include "printf.c"
main() {
        printf("Hello, world!\n");
        }
```

Files PRINTF.C and CLIBRARY.ASM should be copied to A: [SY0: on HDOS]. Then HELLO can be compiled and run by the following steps. Characters which the computer types are underlined in this example; the other characters are typed by the user.

```
A> c hello

C/80 Compiler 2.0 (2/3/82)

A> as hello

8080 AS 2.2 (7/7/81)

A> hello
Hello, world!
A>
```

6. RUNNNING THE COMPILER

The simplest way to compile a C/80 program is to give the command

c filename

This takes the source file FILENAME.C and produces an assembly file FILENAME.ASM. Of course, any file name can be used instead of filename.

In order to create an assembly file with a different name or on a different device, the command looks like

c d:outfile.mac=b:infile

[c dk1:outfile.mac=sy1:infile on HDOS]

If no extension is specified, the defaults are .ASM for the output file and .C for the input.

The C command may include "switches" to select compiler options. The switches consist of a -, a letter (upper and lower case are synonymous), and sometimes a numeric value (represented below by N).

Example: To compile file FOO.C, producing file B:FOO.ASM, including the source text as comments in the assembly file, and allocating space for 400 symbols (300 global and 100 local), use the command

c -t -s400 b:foo=foo

The default values for the switchable parameters are listed on file PATCHES.DOC on the C/80 distribution disk, and may be patched by the user to select different defaults. The switches are:

-t Include the source program text as comments in the assembly language file.

-m Generate Macro-80 assembler output. See Section 10.

-p Generate a runtime profile for the program being compiled. See Section 12.

-sN Allocate N entries in the symbol table. 3/4 of the symbol table entries are used for globals and the remainder for local variables. Memory occupied by the symbol table is about 16 bytes per entry.

-cN Allocates N bytes for the string constant table. This table stores all string constants in the program.

-dN Allocates N bytes for the **#define** table. The define table stores all **#define** macro names and strings.

-wN Allocates N slots for the switch/case table. This table determines the maximum number of **cases** in a **switch** statement or in nested **switch** statements.

-rN Allocates N bytes for the structure table. This table stores the information from structure declarations.

-lN Begin generating internal labels at number N (default 0). This provides a method of compiling several C source files into separate assembly files which may then be assembled together. See Section 11.

-g Do not reserve storage for globals. This is equivalent to preceding each global declaration with the **extern** keyword. This may be useful in multiple file compilations; see Section 11.

-f Normally, the compiler tries not to duplicate strings which can be overlapped. In particular, two identical string constants will point to the same location. This may cause problems in a program which alters the contents of a string constant or an initialized character pointer. (Initialized character arrays are not affected.) The -f switch turns off the string overlap feature, and insures that each string is stored separately.

-z C generally initializes all static and global storage to zeros. Since this can create a very large intermediate assembly language file, C/80 only zeros arrays shorter than 256 bytes. The -z switch causes all statics and globals to be initialized to zero, regardless of size.

After compiling the source program, use AS [ASM on HDOS] to assemble the assembly language file into an executable program. Note that all C programs will include the C runtime library file, CLIBRARY.ASM, which must reside on A: [SY0: on HDOS].

If more than the minimum necessary memory is available, the C/80 compiler may be patched to change the default values to larger, more convenient ones. The locations in C.COM [C.ABS on HDOS] which contain the default values are shown in file PATCHES.DOC on the C/80 distribution disk. The program may be patched on CP/M by using DDT and the SAVE command, and on HDOS by using the PATCH program.

7. C/80 LANGUAGE SUMMARY

This section contains a language summary in tabular form, followed by a concise explanation of the major C language features. Its aim is to convey the basics of C/80 to a programmer with some feel for computer languages. It is not intended to replace The C Programming Language as an exhaustive reference.

(1) Data Types:

Types: **char**, **int**
long (in C/80, same as **int**)

Declarations:

```
char c, *pc, ac[], ac2[n], **x[m][n]
int i, *pi, ai[], aci[n];
extern char/int ...
static char/int ...
auto char/int ...
register char/int ...
initialization:
        char/int v = constant;
        char/int a[] = {c,c,...};
```

Structure Declarations: See "Structures" below.

(2) Primaries:

constant:

- decimal number
- octal number beginning with 0
- hex number beginning with 0x or 0X
- character constant 'c'
- string "abc"

variable

address[expression]

function(argl,...,argn) (n >= 0)

structure.element

ptr_to_structure->element

(3) Expressions:

Unary operators:

-	minus
*	contents of
&	address of
++	increment (pre- or postfix)
--	decrement (pre- or postfix)
!	logical not
~	bitwise not
(type)	type is any type (e.g., char*); forces type of following expression
sizeof	nr. bytes in type or expression

Binary operators:

* /	arithmetic operators
%	modulo
+ -	arithmetic operators
>> <<	right, left shift
< <=	less than, less than or equal
> >=	greater than, greater than or equal
== !=	equal, not equal (0 or 1 valued)
& | ^	bitwise and, or, xor
&& ||	logical and, or
? :	if-then-else expression
=	assignment operator
+= -=	arithmetic assignment operators
*= /=	
&= |=	
^= >>=	
<<=	
,	comma (expression sequencing)

(4) Statements

```
expression;
if (expression) statement;
if (expression) statement; else statement;
for (expression; expression; expression) statement
while (expression) statement;
do statement while (expression);
switch (expression) {
        case: statement; ...
        default: statement ; }
break;
continue;
return;
return expression;
goto label;
label: statement;
{ statement; ...; statement; {
;       (null statement)
```

(5) Function Definitions (functions are fully recursive)

```
fname(arg1,...,argn)
        int/char argi,*argj, ... ;
        { statements; }
```

(6) Preprocessor Functions

/* comments */	
#define name string	Replace name by string throughout text
#undef name	Erase definition
#include "filename" or #include <filename>	Inserts filename at that point.
#ifdef name	Generate following code if name is #defined or
#ifndef name	not #defined, or if
#if expression	expression nonzero, respectively.
#else	Reverse conditional generation
#endif	End conditional generation
#asm	Begin assembly language
#endasm	End assembly language
#UPPER	Convert upper to lower case

7.1. Variables.

Variable names consist of letters, numbers, and the character "_". The first character must be a letter. Upper and lower case letters are allowed and are different (except globals when Macro-80 is used); usually lower case letters are used. Variable names may be any length, but any letters after the first seven are ignored (six for globals with Macro-80).

Each variable has a type, a scope, and a storage class.

The scope of a variable determines the portion of the source program within which the variable is known. The three possible scopes are local, global, and external. Local variables are those declared at the beginning of a function body; they are known only inside that function and the same names can be used in other

functions. Global variables are those declared outside a function body; they are known in all functions from the declaration to the end of the file. In addition, the **extern** declaration may be used to reference variables in the C/80 library or on other files in a multiple file compilation.

7.2. Data Types

C/80 contains two basic data types: **int**, which is a 16 bit signed integer (range -32768 to 32767), and **char**, which is an 8 bit signed integer (range -128 to 127). Chars are often used to store characters of text. Unsigned ints are are also supported.

Integers and characters are the basic components of the more complex data types: arrays, pointers, and structures. The following declaration declares an int, a doubly dimensioned int array, an array of pointers to ints, and a function returning pointers to ints.

```
int i, array[30][10], *ptr[5], *f();
```

An array is similar to a BASIC or FORTRAN array; it simply consists of consecutive pieces of memory each large enough to contain a char, int, or pointer.

C does not contain a separate string data type; strings are stored as arrays of chars. By convention, a byte containing 0 is used to terminate a string.

7.3. Pointers

The concept of pointers is essential to the C language. A pointer is simply an address. Thus, in C/80, pointers are unsigned 16 bit numbers, similar to unsigned ints. They are used to step efficiently through arrays, where other languages might use subscripts and an index variable, and to pass (addresses of) large data structures as function arguments. An indication of how to use pointers is given in Section 7.7.

7.4. Structures

Structures are a useful way to organize data. An example of a structure declaration is:

```
struct tree {
    char value[5];
    struct tree *left,*right; }
                forest[50],*ptree;
```

This declares three kinds of things: a structure type called **tree**, structure elements called **value**, **left** and **right**, and variables: an array of structures, called **forest**, and a pointer to objects of type **struct tree**, called **ptree**. Each object of type structure is like an array. But whereas an array contains a number of pieces of data all

of the same type, a structure contains pieces of data called elements, which may be declared to be of different types. In the declaration above, objects of type **struct tree** are declared to contain a 5-long character array, and two pointers to things of type struct tree.

In a structure declaration you can omit the variables, the structure name, or (once the structure type has been declared) the { ... }.

A structure element can be referred to by the operators -> or .:

```
forest[i].value
ptree->left
```

Such constructs can be treated just the same as variables. Use . to refer to fields of variables which are structures, and -> to refer to fields of things which point to structures.

Structures and their use are a complex topic and can only be touched on here. The C Programming Language contains much discussion and many examples which will be helpful.

7.5. Storage Classes

Storage classes determine how a variable is stored in memory. The storage classes in C/80 are:

```
static
auto
register
extern
```

Statics are simply memory locations. **Auto** variables are stored on the pushdown stack. **Statics** declared within a function are preserved between calls of that function, whereas **autos** are not. When a function is called recursively, a new **auto** is created local to that call of the function; **statics** are not. Globals are always **static**. Local variables default to **auto** but may be declared **static**. (C/80 actually uses static storage for local 16 bit variables and saves the values so that this is transparent to the user but more efficient than using the stack. The explicit **auto** declaration may be used to override this and force the variables to be located on the stack.)

Register variables are stored efficiently, but are otherwise the same as **autos**. **Externs** are **statics** which are declared in another source module.

7.6. Constants.

A decimal constant consists of a string of decimal digits. A constant beginning with '0' (e.g., 0177) is interpreted as an octal number. A constant beginning with '0x' or '0X' is a hex constant.

C/80 computes constant expressions at compile time. That means that wherever an integer constant is required (such as the dimension in an array declaration), you can use an expression containing only constants.

C/80 also contains string and character constants. Characters are any printing character, or

```
\t      for tab
\n      for newline (end of line)
\r      for carriage return
\b      for backspace
\f      for form feed
\\      for \
\'      for '
\"      for "
\123    for the octal value 123
          (or any other value)
```

A single character constant is written 'c'. A string constant is written "ccc...". A string constant is stored as a 0-terminated array of **chars**. Useful things to do with string constants include assigning them to **char** pointers, and passing them as arguments to functions.

7.7. Operators and Expressions.

The operators in C are shown in the table at the beginning of Section 7. They appear in the approximate order in which they are performed during expression evaluation; e.g., / is performed before + is performed before &.

Certain operators are peculiar to C. The unary operator * takes a pointer and yields the contents of the location it points to. The operator & takes an object, which must have an address, and yields a pointer to that address. Thus, for any expression A which has an address, the value of *&A is the same as A.

The operators ++ and -- can appear either before or after a variable. They cause the variable to be incremented (++) or decremented (--) by 1. The value of the expression is the variable either before (V++) or after (++V) the operation. For example, if p is a pointer to char, then *p++ increments p, but applies the * operation to the value of p++, which is the value of p before being incremented. This leads to the following sort of code, which is common in C:

```
p = "Any old string\n";
while (*p) putchar(*p++);
```

which outputs the string by calling **putchar()** with each character in it, until the 0 byte terminating the string is encountered.

Note that when a pointer is incremented or decremented, its value changes not by 1 but by 1 object. Thus incrementing a pointer to an **int** moves it to point to the next following **int**; its actual value increases by 2. A pointer to a structure is incremented by the size of the structure, in bytes.

Truth values in C are either zero (false) or nonzero (true). Truth-valued operators (==, >, &&, etc.) return 1 for true. A useful expression in C is

```
expr ? tvalue : fvalue
```

whose value is tvalue if expr is nonzero, and fvalue if expr is zero.

7.8. Statements.

The table at the beginning of Section 7 lists the statements in C. Anywhere that C allows a single statement, it will also accept a compound statement of the form

```
{ statement; statement; ...; }
```

The iterative statements in C all have simple equivalent definitions:

```
for (e1; e2; e3) statement;

        means       e1; L: if (e2) { statement; e3; goto L; }

while (e) statement;

        means       L: if (e) { statement; goto L; }

do statement; while (e);

        means       L: statement; if (e) goto L;

switch (e) { case c1: s1; ... case cn: sn; default: s; }

        means       if (e == c1) s1;
                    :
                    if (e == cn) sn;
                    s;
```

The **switch** statement is not exactly equivalent to what is shown, however. The expression e is actually evaluated only once. The **case** values c1,...,cn must be constants. And the **default** case may be eliminated, in which case no **case** is executed if the value e is different from the values c1,...,cn.

The statement **break** jumps out of the smallest for, while, or switch containing it. The statement **continue** begins the next

iteration of the smallest for or while containing it.

7.9. Conclusion.

This short a summary can not begin to present all the details of the C language. In order to learn more, you can look at the sample C programs provided on the C/80 distribution disk and read The C Programming Language.

8. IMPLEMENTATION AND MACHINE DEPENDENCIES

The reference manual for C/80 is The C Programming Language (see Section 2). In using that book, it must be kept in mind that some features of C and the runtime library are not present in C/80, or differ from the description in the book. Those omissions and differences are listed here. First the unimplemented features and major restrictions are listed, followed by a detailed listing of all differences. Section numbers refer to the C Reference Manual in Appendix A of The C Programming Language.

Unimplemented Features

Float, **double**, **entry** and **typedef** keywords (2.3)
Long and **float** constants (2.4) and arithmetic
Typedef (8.1, 8.8)
Bit fields (8.5)
#line (12.4)

Major Restricted Features

Function calls (7.1) must have the same number of arguments as the called function definition.
Blocks (9.2): declarations are allowed only at start of a function.
#define (12.1): arguments are not allowed.

Complete Difference List

2 Lexical conventions

Blanks in the middle of multiple character operators (e.g., **=***) are not allowed.

2.2 Names

The first seven characters of a name are significant. If Macro-80 is used, global symbols are restricted to 6 characters, and upper and lower case are the same in globals.

2.3 Keywords

Float, **double**, **entry** and **typedef** keywords are not recognized.

2.4 Constants

Long and floating constants are not implemented.

2.6 Hardware characteristics

Char is 8 bits. Int, short and long are 16 bits.

7.1 Primary expressions.

Function calls must have the same number of arguments as the called function definition.

7.13 Conditional operator.

In a?b:c the type of the result is that of b if b is a pointer, and the type of c otherwise. (See also Sect. 15 below.)

8.1 Storage class specifiers

Register variables must be at most 16 bits long. They are stored in static memory for fast access, and are saved on function entry and exit. C/80 allows any number of register variables, and the & operator may be applied to them, but such code may not be portable.

Auto variables are stored on the stack. Local variables default to register if they are 16 bits long, but the **auto** declaration can override this. Other local variables, and all arguments, default to auto, but the register declaration can be used to override this (except for variables longer than 16 bits). All this is transparent to the user.

The scope of an **extern** declaration is the remainder of the source file, even if the declaration is within a function definition.

Typedef is not recognized.

8.5 Structure and union declarations

Bit fields are not implemented.

8.6 Initialization

Only static and global variables may be initialized. Only objects smaller than 256 bytes are defaulted to 0; the -z compiler switch removes this restriction. (See also Sect. 15 below.)

Type declarations can not be nested. About the only restriction this imposes is that **sizeof** a type name may not be used in a dimension in a declaration.

8.8 Typedef

Typedef is not implemented.

9.2 Compound statement, or block

Declarations are allowed only at the beginning of a function body.

12 Compiler control lines

In-line assembly language is supported by the **#asm** and #endasm directives.

12.1 Token replacement

#Define is not applied recursively, and arguments to macros are not allowed.

12.4 Line control

#line is not implemented.

15 Constant expressions

?: and & are not allowed in constant expressions, except that & may be the first character in an initializer.

17 Anachronisms

All forms listed are recognized, except that initializers which lack an = and start with (name... will not compile.

Printf and fprintf:

These functions (and **sprintf**) cope with a variable number of arguments through use of a **#define** kludge which redefines **printf**. This requires that either **#include "printf.c"** or **#include "printf.h"** be placed before the first use of **printf**. Otherwise, the C/80 1.6 **printf** and **format** must be used: see Section 9.6.

I/O and Runtime Library:

Many of the basic library functions and treatments of files described in the manual are supported, although the format is not always identical. **Getchar, putchar, getc, putc, fopen, fclose, seek, ftell and exec** are provided. **EOF** has the value -1; **NULL** is 0. The older Version 6 convention of **fin** and **fout** is followed instead of **stdin, stdout** and **stderr**. I/O redirection is provided. The I/O and runtime library is described more fully in the next section.

9. RUNTIME AND I/O LIBRARY

C/80 is supplied with a runtime library, which provides convenient access to files and other devices in a manner generally consistent with accepted C conventions.

9.1. Files and Devices:

NOTE: The following information applies only to use of C/80 on the CP/M operating system, and should be ignored by users of HDOS.

Under CP/M, the C/80 library recognizes the logical device driver names CON:, RDR:, PUN: and LST: as legal file names.

In doing I/O to CON:, C/80 normally uses line at a time mode. This is true whether CON: has been accessed explicitly by opening file "CON:", or as the default device for getchar and putchar. If you need to use character at a time console I/O, declare

```
extern char Cmode;
```

and set **Cmode** to zero. (It is initially set to 1; other values produce undefined results.)

When reading from CON:, in addition to the control characters which are interpreted by CP/M (see the description of functions 1 and 10 in the CP/M 2.2 Interface Guide), C/80 interprets ctrl-B as an interrupt. Moreover, when doing I/O to any of the four logical devices, C/80 maps CR into '\n' on input and '\n' into CR-LF on output. Calling **fopen** to open the device in binary mode has no effect on this (although it does for file I/O); the only way to avoid these mappings is not to go through the C/80 library.

[Under HDOS, any legal file or device name may be used whenever a file name is called for. Examples of legal names are FOO, SY1:FILE.DAT, or LP:.]

Upper and lower case letters are legal and synonymous in file names. On CP/M, the user is responsible for insuring that characters which confuse the CCP, such as '.' and ':', are not used.

9.2. Commands:

C/80 programs begin execution by calling the routine **main**, which the user must provide. **Main** should start off with the declaration

```
main(argc,argv)
char *argv[];
```

When **main** is called, **argc** will be the number of elements in **argv**, and **argv** will be an array of pointers to the strings which appear on the command line (except for **argv[0]**, which may not contain anything useful.) **Argv[argc]** is always -1. For example, if a C/80 program named **progl** is run with the command

progl a b foodle

then **argc** is 4, and **argv** contains pointers to the strings "PROG1" (or nonsense), "A", "B" and "FOODLE" (e.g., **argv[3]** is "FOODLE"). An argument containing spaces and/or tabs may be enclosed in either single or double quotes.

Due to limitations in both the HDOS and CP/M systems, lower case letters in the argument line are passed to **main** as the upper case equivalent.

9.3. I/O Redirection:

Many C programs do input and output a character at a time, taking input from the standard input using the routine **getchar**, and writing to the standard output using the routines **putchar**, or **printf**. The standard input and standard output are both initially the terminal. However, files or other devices can also be used as the standard input and/or output.

This is implemented by two global extern ints, **fin** and **fout**. These variables define the standard input and output, respectively. They are usually set to 0 before main is called, and I/O is done to and from the terminal. However, a program may open a file for reading or writing, and set **fin** or **fout** to the file's channel number. This will cause subsequent I/O to be done to the file rather than to the terminal. A device may be used instead of a file.

Normally, an output file must be closed explicitly by a program before exiting. However, the I/O library always closes **fout** when a program is terminated (including abnormal termination by ctrl-C under HDOS and ctrl-B under CP/M).

Fin and **fout** may also be redefined in the command line when the program is run. For example, the command

progl a b <b:infile >lst:

will run progl with arguments "A" and "B". **Getchar()** will read characters from file "B:INFILE", and **putchar()** will write characters to the printer. The > and < arguments are not included in **argc** or **argv**, and may appear anywhere in the argument list. [The equivalent HDOS command would be

progl a b <syl:infile >lp:]

Since the C/80 compiler is itself a C/80 program, I/O redirection may be used to redirect to a file the error messages usually output to the terminal.

9.4. Interrupting a Program:

C/80 programs may be interrupted by ctrl-B or ctrl-C [ctrl-C only on HDOS], which causes the program to terminate immediately and exit. The standard output is closed (except by ctrl-C on CP/M), but all other open output files are lost.

NOTE (applies to CP/M users only): Under CP/M, a ctrl-C may not always be noticed when it is typed. The C/80 library will check for a ctrl-C whenever a disk read or write is performed, but if a program does not access disk it must take special steps in order to be interruptable.

The HDOS operating system provides a way for the user to trap and handle interrupts caused by ctrl-A, ctrl-B and ctrl-C. C/80 provides a similar, but weaker, capability under CP/M. When the C/80 console input routine detects a ctrl-B, a call is executed to the location contained in **CtlB**. That location initially contains **exit()**, and thus ctrl-B usually aborts a running program, just as ctrl-C does.

Instead of exiting, your program may choose to handle ctrl-B interrupts itself. To do so, declare **CtlB** to be an **extern int**, and set it to a subroutine to be executed when ctrl-B is typed. If that subroutine returns when done, program execution will continue.

Since CP/M does not usually detect a typed character, even ctrl-C, until the program attempts to read from the keyboard, C/80 programs will often not respond immediately to a ctrl-B or ctrl-C. The C/80 library does check for these characters whenever a disk read or write is performed. You can make your program check more often by calling **CtlCk()** whenever a check is desired.

One side effect of **CtlCk()** is to read any typed character that may be waiting at the console. If the character is not a ctrl-B or ctrl-C, it will be echoed and will be placed in the input line buffer for the next call to **getchar()**. So calling **CtlCk()** provides a limited typeahead capability. However, editing characters like ctrl-U and DEL do not operate on characters read in this manner. Also note that once the buffer has been filled (about 130 characters), any further typed characters will be echoed but discarded without warning, until the buffer has been completely emptied.

9.5. I/O Library Routines:

The following standard I/O library routines are included in C/80 and may be called from any C/80 program:

getchar() - returns a character from the standard input (usually the terminal). -1 is returned for end of file (ctrl-D under HDOS; ctrl-Z under CP/M).

putchar(c) - writes the character c on the standard output (usually the terminal), and returns c.

fopen(fname,mode) - opens the named file and returns the channel number of the file. **Fname** is a string constant, or pointer to or array of characters containing the file name. The name may be any legal file or device, like SY1:FOO.TXT, or TT: (under HDOS; CP/M equivalents are B:FOO.TXT and CON:). **Mode** may be "r", "w" or "u" for read, write or update mode. (Update mode is treated the same as write mode but the file is not deleted before opening. [On HDOS, it is opened in update mode.] **Mode** may also be "rb", "wb" or "ub" if the file is to be treated as a binary file (see **getc**). If the file can not be opened, **fopen** returns 0. A file or device which is written on must be closed explicitly, or the entire file will be lost (except for file **fout**).

At most 6 files may be open at any one time. I/O buffers are allocated for three files. If more than three files are opened, **fopen** will call **alloc** to allocate a buffer of 256 bytes for each additional file. If there is not enough memory available, the open may fail.

[HDOS: The **fopen** channel number is not necessarily the same as system's channel.]

getc(chan) - returns the next character from the file or device open for reading on channel **chan.** Returns -1 to signify end of file. If the file was opened in binary mode, **getc** will read every byte in the file. Files opened in normal mode are treated as ASCII files. Under CP/M, ctrl-Z is interpreted as end of file, and newlines, which are a CR-LF pair in the file, are read in as the single character '\n'. [Under HDOS, the only special treatment is that 0 bytes are ignored.]

putc(c,chan) - writes the character **c** on the file or device opened for writing or update on channel **chan.** In ASCII files, the conversions listed under **getc** are performed in reverse.

fclose(chan) - closes the file or device opened on channel **chan.** If a file opened for writing is not closed before the program terminates, the last block of the file may be lost. Once a channel has been closed, another file may be opened without exceeding the open file limit (see **fopen**).

read(chan,addr,n) - reads up to **n** bytes from channel **chan** into memory starting at address (pointer) **addr. N** must be a multiple of 256. **Getc** reads one character at a time; **read** provides an alternative method for reading many characters at once. **Read** returns the number of bytes read, which may be less than **n** if the end of file was encountered. **Read** returns 0 if an error occurred. **Read** does not perform any character conversion regardless of whether the file was opened in binary mode. **Read** and **getc** should not both be

used on the same channel.

write(chan,addr,n) - writes n bytes from address **addr** to the file open on channel **chan.** **N** must be a multiple of 256. **Write** provides an alternative to **putc** for outputting many characters at a time. **Write** returns 0 if an error occurred, and the number of bytes written otherwise. **Write** and **putc** should not both be used on the same channel.

exit() - terminates the program and returns to command level. Does not close any open files except the standard output. Returning from **main** has the same effect as **exit()**.

alloc(n) - allocates a block of **n** bytes of memory, returning the address of the first byte, or -1 if that much memory is not available. There is no way to release memory once it is allocated. The alloc area grows upward from the end of the user program, and the stack grows downward from high memory. Although **alloc** will always leave about 500 bytes for stack expansion, it is still possible for the stack to grow into alloc memory (or static storage or program memory, for that matter), with undefined results. Note that **fopen** may call **alloc** to allocate I/O buffers.

9.6. Formatted Output

Formatted output is provided by **printf**, **fprintf** and **sprintf**, which are on file **printf.c** and may be included in a compilation by placing the command **#include "printf.c"** before the first reference to **printf.** If **printf** is included in any other way (e.g., by **XTEXT PRINTF.ASM** or by linking **printf.rel** using Link-80), the header file **printf.h** must be included instead.

These routines provide functions similar to the routines described in The C Programming Language.

printf(stg,v1,...,vn) - prints the values **v1** through **vn** (n >= 0) on the standard output (usually the terminal), using **stg** as the format specification. The characters in **stg** are printed on the standard output, except for conversions of the form %C or %nC, where n is an optional field width and C is a conversion letter. Each conversion takes the next value from the argument list and prints it according to the conversion letter. The conversion letters allowed are:

%d (decimal number output, signed)
%o (octal, unsigned)
%c (single character)
%s (string)
%x (hexadecimal)

fprintf(chan,stg) - like **printf**, but output goes to the file opened on channel **chan** rather than to the standard output.

sprintf(addr,stg) - like **printf**, but output goes into the character array beginning at memory location **addr**.

For example, the program fragment

```
printf("i = %d, s = %s\n",27,"Hi there!");
```

would print out the line

```
i = 27, s = Hi there!
```

Older versions of C/80 used two routines, **printf** and **format**, to perform formatted I/O. If programs using this form of output are to be compiled under this version of C/80, the **#include "printf.c"** must be placed after all references to printf, and **printf.h** must not be used.

9.7. Random Access File I/O:

The file SEEK.C on your C/80 distribution disk contains routines affording random access file I/O capability. This file can be included in your C/80 program by the statement

```
#include "seek.c"
```

(Note: CP/M 1.4 and earlier CP/M releases do not support random file I/O, and these routines will not work on those versions of CP/M.)

SEEK.C contains the following library routines:

seek(chan,offset,type) - moves to a specified position in the file which is open on channel **chan**. The next **getc** or **putc** call will read or write starting at the new location. The value **offset**, which may be positive or negative, specifies the number of bytes that the read/write pointer is to be placed from:

type = 0: the beginning of the file.
type = 1: the current read/write location.
type = 2: the end of the file.

For example, **seek(chan,0,2)** will position the read/write pointer at the end of the file. If **type** = 3, 4 or 5, the pointer is moved **offset** records (256 bytes) instead. **Seek** returns a value of **-1** if an error occurs, 0 for success.

ftell(chan) - returns the current read/write pointer for the file open on channel **chan**. This pointer is the number of bytes before the current position in the file. If the current position is greater than 65535, the value returned will be the offset mod 256; i.e., the byte position in the current record.

ftellr(chan) - returns the current read/write pointer for the file open on channel **chan**, divided by 256.

Using **seek**, it is possible to alter or append to an existing file by opening it in update mode. It may be possible to both read and write to a file, as long as a seek intervenes, but this has not been tested thoroughly. It is safer to close the file and reopen it in the other mode.

9.8. Program Chaining

The file EXEC.ASM [HDOS: EXEC.C] contains a routine which allows execution of another program from within a C/80 program.

exec(prog,args) - chain to another program. **Prog** is a string containing the name of a program. **Args** is a string containing any arguments, separated by blanks, just as on the command line. **Exec** will execute the named program with the arguments given, just as if it had been invoked from the command line. Unless an error occurs, control never returns from **exec**. All open files must be explicitly closed before calling **exec**, or strange things may happen.

10. USING C/80 WITH MACRO-80

C/80 can optionally generate assembly code for input to the Microsoft Macro-80 relocatable assembler. This allows you to develop a C/80 program in several modules, generating .REL files which can be linked using the Link-80 linking loader. You can create .REL modules for C/80 library routines such as PRINTF and SEEK, to speed up their inclusion in your programs. A library manager such as LIB-80 (CP/M versions) can be used to create a library of your commonly used subroutines which can be accessed using Link-80.

To generate a .MAC file, invoke C/80 using the -m switch:

```
c -m [other args ...]
```

You can also patch the Macro-80 compatibility flag so that this mode is the default; the -m switch will then generate .ASM files. The patch location is given in file PATCHES.DOC on your C/80 distribution disk.

Macro-80 files generated by C/80 must be assembled by M80 to create a .REL file. REL files are then linked by L80 to create a .COM file [HDOS: .ABS file]. IMPORTANT: When linking C/80 .REL files, the file CLIBRARY.REL must be linked in and it must be the last file linked.

Using Macro-80 imposes a few additional restrictions on source programs. In global variables and function names, only the first six characters are significant (as opposed to 7 in .ASM files). Also, upper and lower case are considered identical by Macro-80. Global arrays and variables must appear in only one source module, and must be declared extern by any other module that references them, being careful to distinguish between arrays and pointers. Functions are implicitly extern and need not be declared.

11. MULTIPLE ASSEMBLIES

To reduce space and time taken by compilations, it is often useful to compile a large program in several pieces. This may be done by splitting the program into several C source files, using the **extern** declaration to reference globals which are declared in another source file. For example, if the declaration **int i[5];** appears at the top (global) level in one source file, the declaration **extern int i[5];** in a second source file will allow programs in the second file to refer to the array **i** defined in the first file. Simply saying **int i[5]** in both files will cause i to be doubly defined at assembly or load time. Exception: functions defined in one source file may be called from another source file in the same assembly or load without any special declaration.

There are two ways to generate a single object module from multiple compilations. One way is to use Macro-80 and Link-80 from Microsoft, as described in the previous section. The other is to use the AS assembler provided with C/80 [HDOS: use Heath's ASM] to assemble one module from several .ASM files.

An example will illustrate how to do this. Suppose there are three C source files: MAIN.C, SUB1.C, and SUB2.C. First you must insert into MAIN.C statements which will cause the assembly language files for SUB1 and SUB2 to be included in the assembly. The following statements should be put into MAIN.C somewhere at top level (i.e., not inside a subroutine):

```
#asm
        XTEXT SUB1.ASM
        XTEXT SUB2.ASM
#endasm
```

The XTEXT command will cause these lines to be included in the assembly language file MAIN.ASM.

Next, compile MAIN.C to create an output file called MAIN.ASM. This file contains compiler-generated labels, which usually look like .a, .b, and so on.

Then compile SUB1.C to produce an output file named SUB1.ASM. If nothing is done to prevent it, the compiler will use the same labels .a, .b, etc., in SUB1.ASM, and there will be a conflict when the two files are assembled together. To prevent

this, SUB1.C should be compiled with a command like "C -L1000", which will start the labels 1000 down in the sequence. (The largest permissible value for the -L switch is 32767). The -L switch also suppresses the generation of instructions in SUB1.ASM to include the C I/O library, since MAIN.ASM already has those instructions.

Similarly, compile SUB2.C, say by "C -L2000", to produce SUB2.ASM. If disk space is a problem, these compilations may all be performed on different disks, and the .ASM files copied to another disk for assembly. The library file CLIBRARY.ASM must reside on A: [SY0: on HDOS] during assembly.

To assemble the program, run AS [ASM on HDOS] and give the command MAIN=MAIN. All the files will be assembled to produce MAIN.COM [MAIN.ABS on HDOS] which can then be run.

12. RUNTIME TRACE AND EXECUTION PROFILE

Most programs which take very long to run spend most of their execution time executing a relatively small amount of code. C/80 contains a runtime execution profile feature to help identify where a program is spending its time, so the critical routines can be made more efficient.

To use this feature, compile the program with the -p switch, as in

```
c -p progname
```

If you are using AS or ASM, make sure the file CPROF.ASM, from the C/80 distribution disk, is on A: [SY0: on HDOS] when the program is assembled. (CPROF.ASM is not on the Osborne distribution disk, due to space restrictions. You will have to generate it from CPROF.C by the command

```
c -L32600 cprof )
```

If you are using Macro-80, compile and assemble CPROF.C to produce CPROF.REL, and link it in when you load.

Now run the program. When the program finishes running and exits normally, a listing will be produced on the standard output device showing, for each subroutine, the number of times it was called and the total time (in units of two ticks of the computer clock) spent inside the subroutine.

Similarly, a runtime trace of the program execution can be produced by copying file CTRACE.ASM from the distribution disk onto A: [SY0: on HDOS] as file CPROF.ASM, and compiling as above.

> Note (applies to CP/M users only): C/80 can only provide execution times if your system has a 16-bit clock at some address in memory. The Heath/Zenith systems have a clock

at memory location 0B hex. To use the profile feature on systems with no clock, or with a clock at another location, you must regenerate file CPROF.ASM as follows:

Edit file CPROF.C. Locate the line which begins with **#define TICCNT**. If your system has a clock, replace the expression following **TICCNT** with the address of the clock word in memory. If there is no clock, comment out the entire line. Then recompile CPROF by the command

c -L32600 cprof

13. THE AS ASSEMBLER (CP/M Versions Only)

CP/M versions of C/80 come with AS.COM, an absolute 8080 assembler. AS is essentially the same as the ASM assembler under Heath HDOS. This section gives a brief description of AS, to help you write assembly language code to be included in C/80 programs.

To assemble a program, type a command of the form

as comfile,listfile=infile

where comfile is the name of the absolute file to be produced, listfile is the file on which to write an assembly listing, and infile is the assembler source file. If no extensions are specified, they default to .COM, .LST and .ASM respectively. **Listfile** can be a C/80 device, such as LST:; if that device does not respond to tabs, it is more useful to list to CON: and use ctrl-P to obtain a hard copy. The command

as filename

is short for

as filename=filename

AS takes Intel 8080 mnemonics, upper case only. Identifiers are up to 7 characters from the set A-Z, a-z, 0-9, ., _ and $. Upper and lower case letters are distinguished in identifiers. A label may be followed by one or more colons, which are ignored.

Constants can be one or two characters enclosed in single quotes, or a string of digits, possibly with a suffix O or Q for octal, H for hex, or B for binary. Default is decimal.

The symbols * and $ represent the address of the current instruction.

Arithmetic expressions in an address field are evaluated strictly left to right. The operators are +, -, *, /, & (bitwise and), and < (left shift). Parentheses are not allowed.

The following pseudo-ops are identical to the ones in the CP/M ASM assembler: ORG, EQU, DB, DW, DS.

The pseudo-op "XTEXT filename" is the AS equivalent of the C/80 "include" preprocessor directive. It includes the named file at that point in the assembly. If no disk is specified, it assumes the disk on which the current source file resides.

The pseudo-ops "LON ccc" and "LOF ccc" control listing. "ccc" is a string of characters: L turns listing on (LON) or off (LOF), C controls listing of lines from XTEXT files (default off), and G lists all bytes generated by an instruction (default: list just the first five).

14. UPPER CASE SOURCE FILES

Since the C/80 language depends heavily on the full ASCII character set and on lower case keywords, a facility is provided to allow C/80 source files to be prepared on upper case only terminals. Upper case source files should contain the preprocessor command

#UPPER

as the first line in the file. This causes the compiler to interpret the remainder of the file in upper case mode. In this mode, the compiler translates each upper case letter to the corresponding lower case letter. Upper case letters, and the special characters not available in the upper case ASCII subset, are typed as follows:

The characters:	are interpreted as:
\(	{
\)	}
\!	|
\-	~
_	^
A ... Z	a ... z
^A ... ^Z	A ... Z

In upper case mode, when the character ^ appears before a non-alphabetic character, it is ignored. Note that ^ may be displayed on some terminals and printers as an up arrow, and on others as a caret or "hat".

The **#UPPER** command affects only the file in which it appear, and does not have any effect on an **#include** file. If an **#include** file is prepared in upper case mode, it must contain its own **#UPPER** command. (Thus, PRINTF.C, which is not in upper case mode, may be included in a file which is in upper case mode.)

The file UCASE.C, which is included in the C/80 distribution, is an example of an uppper case mode source file.

15. TRICKS AND INTERNALS

Global arrays occupy space in the .COM [or .ABS] file. To reduce the size of these files, it is best to allocate large arrays at run time. This can be done by making them local to a function, or by using **alloc()**.

Internal assembly language routines invoked by C function calls have access to all registers and need not restore their values.

If you want to fetch the value of a variable using in-line assembly language code, put a statement consisting only of the variable just before the **#asm** directive. C/80 will leave the variable value in HL.

The C function call linkage is to push the arguments, as 16-bit quantities, on the stack, and call the subroutine. It is the responsibility of the calling program to pop the arguments off the stack on return. Note that standard C pushes its arguments in reverse order; this means a C/80 routine may not usefully be called with a different number of arguments than it expects.

The compiler generates code using a primary register (HL), a secondary register (DE), and the stack. Most operations are performed by calling library subroutines, with the first operand being HL and the second operand either DE or the top of the stack (which is usually popped).

In order to produce code which is both space and time efficient, variables should be declared **static** whenever possible, and **int** in preference to **char**. Declaring a function argument to be **register** will save space if the argument is used five or more times in the body of the function.

When compiling using several source modules, it is handy to define global variables using an included header file. But these variables must be defined only in one module, and declared **extern** in all the others. This can be accomplished through the following programming trick:

In the header file, declare all the variables as follows:

```
#ifndef EXTERN
#define EXTERN extern
#endif
EXTERN int i, j, ...
```

Then, in the one file in which the variables are to be defined, place

```
#define EXTERN
#include "header-file"
```

At runtime, the library allocates three I/O buffers directly below the operating system area, and then builds the stack downward

starting below the buffers.

Note (applies to HDOS users only): Under HDOS, device drivers may load below the overlay area when the device is opened and closed. If a device driver spills over the 768 byte buffer area and into the stack, problems may occur. Since HDOS provides no satisfactory method for determining potential driver memory requirements, the runtime library can not anticipate this problem automatically.

The problem will rarely arise, since most systems have at most a single LP: or AT: device (in addition to TT: and SY:, which are always loaded), and no single driver is large enough at present to cause trouble. If multiple device drivers are to be loaded it may be necessary to leave additional room for the drivers. To do this, increase the number 10 in the LXI B,10 instruction in CLIBRARY.ASM (around line 40), adding the number of bytes necessary (1000 per extra device driver should be generous).

16. COMPILER ERROR MESSAGES

When a C/80 program contains a detectable error, the compiler will produce an error message, giving the source file name and line number, and a description of what the compiler thinks the error is. The source line is listed, with an arrow pointing to the location of the error in the line. (The line is shown the way it looks after all #defines have been expanded.)

These messages aren't always as helpful as one would like, however. For instance, the compiler may be looking for a different statement type than the one you thought you wrote. So the message may not describe the error. Sometimes, the compiler detects the error far away from the place where it actually occurred. For example, leaving off a '}' deep inside a function will cause the next to last '}' in the function definition to terminate it, at which point the compiler will probably spew out dozens of error messages as it tries to parse executable statements as declarations. To find such an error, you may have to inspect many lines of code.

In addition, once an error has been detected, the compiler is not always able to recover and continue parsing the remainder of the program. So a single error will sometimes result in a large number of error messages. Often, this is the result of C/80 trying to parse statements as declarations or declarations as statements. When you can't understand what some of these messages are complaining about, you should fix the first error detected, and any others you can easily locate, and then recompile to find any remaining ones.

Limitations on error detection and analysis are almost unavoidable in compilers, especially when trying to cram a powerful

language into the limited space of a microcomputer. This section can help by describing C/80's error messages and what might cause them to occur.

bad label: Labels must follow the rules for identifier names.

bad type: C/80 expected to see a type: **short, long, char, int, unsigned, struct** or **union.**

can't subscript: You can only subscript a pointer or an array. (Did you declare it?)

can't initialize union: Like it says.

can't find file: Did you specify the file and extension correctly? Did you leave off the device? Also, when available memory is almost exhausted, the compiler may be unable to allocate the buffer space to open an **#include** file.

can't compute size: Maybe the arrays in the expression have been declared incorrectly.

construct not permitted: C/80 does not allow nested type declarations. For example, **sizeof** an abstract type can not be used as a dimension in a declaration.

dimension missing: The only time an array dimension can be omitted (or 0) is when declaring a function argument, or when the dimension is determined by the size of an initializer. Furthermore, only the last dimension of an array can ever be omitted.

extra ; (ignored): This looks like a function declaration, except function declarations are **f(...) { ...}** and you put a ; after the). So the compiler took it out.

ifdefs nested too deep: Maximum nesting of **#ifdef, #ifndef** and **#if** is 5.

illegal struct reference: Either a . preceded by something that isn't a structure, or a . or -> followed by something that isn't a structure element. Remember that . is used following things that <u>are</u> structures, and -> following things that <u>point to</u> structures.

illegal function call: The identifier or expression is not of type function.

illegal expression - need lvalue: Some operations (& and assignment, for example) require an lvalue, which is an object with a memory address. This isn't one. (Things that aren't lvalues include constants and expressions.)

illegal symbol name: The compiler wanted an identifier here.

improper argument: This argument is too large; probably a structure

or union. Try using a pointer to it instead.

internal compiler error: The compiler encountered an error in code generation. This may indicate a compiler bug; please report it to The Software Toolworks. You may be able to proceed by simplifying or rearranging the expression.

invalid expression: The compiler is looking for an expression, but this does not look like one.

line too long: Line longer than 100 characters. This error can be caused when **#define** expansions make a line longer.

macro table full: Recompile using the -d switch to increase the size of the **#define** table; see Section 6.

misplaced case: **case** not inside a **switch** statement.

? missing: (where ? is some punctuation character): C/80 expected to find that character and didn't. It inserted it and continued, so if you really did leave that character out at that spot, the compilation proceeded correctly.

must be a constant: The compiler looked for a constant or constant expression, but didn't find it. Remember that **?:** is not legal in a C/80 constant expression.

nested too deep: **#include** files can be nested no more than three deep (a main file and two included ones).

no active whiles: A **break** or **continue** statement was not inside any **for**, **while** or **switch** statement.

not a label: This construct needs an identifier which is a label. You have used something which is either not an identifier or something besides a label (like a char or int).

not a declared variable: The compiler wants an lvalue here. (See "illegal expression - need lvalue" above.) Usually this error means you have forgotten to declare an identifier; all identifiers in C must be declared before use.

not a function: This identifier or expression is followed by a '(', so it looks like a function call. But it's not of type function.

not an address: In an initializer, you applied **&** to something that does not have an address.

not a pointer: You applied ***** to something that is not a pointer. (Maybe you applied it implicitly, like in a subscript operation.)

operands and/or operator incompatible: This usually means you have performed an illegal arithmetic operation on a pointer. The only legal ones are pointer plus integer, and pointer minus either an integer, or another pointer that points to

something of the same size. If the expression looks legal, try rearranging the order of the operands. If you know what you are doing but C won't let you (like computing pointer & mask), use the **(int)** cast to fake the compiler out.

output file error: Usually means the disk is full. You won't see this for long, because the compiler will start dumping the assembly language output to the terminal.

previously defined: This identifier has been defined before.

string space exhausted: Recompile using the -c flag to increase the size of the string table (see Section 6).

struct table overflow: Recompile using the -r flag to increase the size of the structure table (see Section 6).

symbol table overflow: Recompile using the -s flag to increase the number of symbol table slots allocated (see Section 6).

syntax error: The compiler was trying to find a declaration (at the top level) but couldn't.

too complex: In an initializer, this expression was too complicated, or was not a constant.

too complicated: A type declaration had more than 7 levels of indirection (*, [] or ()).

too large for register: Register declarations may only be applied to chars, ints and pointers.

too many active whiles: Well, congratulations. There's only one table in this compiler that isn't expandable, and you have overflowed it by nesting 20 **whiles**, **fors** and/or **switches**. Simplify your program.

too many cases: Recompile using the -w switch to expand the switch case table.

too many structs: At most 239 different struct types may be declared.

type mismatch: You tried to initialize an identifier with a value of the wrong type.

undefined struct name: This struct type has not been declared.

usage error: Indicates a violation of some C usage rule, like passing a struct as a function argument.

warning: =? op assumed: You are using the old style assignment operator (like =&), and failed to leave a space between the operator and the following operand. The compiler assumes you mean =&, but wants you to make sure.

INDEX